# Boot Loader 6

## 6.1 INTRODUCTION

The Boot Loader is a utility program supplied with the ADSP-21000 Family Development Software. The loader converts an ADSP-21xxx executable program, generated by the linker, into a format which can be used to boot a target hardware system and initialize its memory. The loader's invocation command is LDR21K.

The boot loader replaces the MEM21K memory initializer used with the ADSP-21020 processor. Any executable file to be processed with the LDR21K boot loader *must not* be processed by MEM21K. The `-nomem` switch of the C compiler should be used when compiling any C source files—this switch prevents the compiler from running MEM21K.

The following naming conventions are used throughout this chapter:

The *loader* refers to `LDR21K` contained in the software release.

The *boot loader* refers to the executable file that performs the memory initialization on the target.

The *boot file* refers to the output of the loader that contains the boot loader and the formatted system configurations.

*Booting* refers to the process of loading the boot loader, initialization system memory, and starting the application on the target.

*Memory* is referred to as being either **data memory**, **program memory**, or **internal memory**. Remember that the ADSP-21020 processor has separate external program and data memories, but does not have any internal memory. The ADSP-2106x SHARC has both internal and external memory.

# 6 Boot Loader

To use the loader, you should be familiar with the hardware requirements of booting an ADSP-21000 target. See Chapter 11 of the *ADSP-2106x SHARC User's Manual* or Chapter 9 of the *ADSP-21020 User's Manual* for further information.

Different hardware architectures can be created with ADSP-21000 Family processors. In some systems, the ADSP-21000 processor executes a program directly from ROM. For systems that use the ADSP-21020 or that use the ADSP-2106x processor configured to execute its program directly from 48-bit wide external ROM, the PROM Splitter utility (`SPL21K`) is normally used instead of the loader to create ROM-based boot files. In systems that use 8-bit wide boot PROMs, the loader utility configures the executable code to be downloadable through this narrower datapath. It does this by creating a small executable, the boot loader, which is added to the assembled / compiled source files, and is downloaded first into the processor at reset. The boot loader then downloads the actual program to be run on the system, initializes memory (both internal and external), and finishes up by copying over itself and jumping to the restart vector address. The boot loader is designed to have a minimal impact on the target executable.

The loader supports 32-bit, 40-bit and 48-bit data memory. It is designed for use with systems that execute from RAM, either internal, external, or both.

The target application's **architecture file** is important to the loader. It specifies the type of processor in the target and its memory configuration, especially the widths of the memory segments. The loader uses this information to initialize memory correctly.

The ADSP-2106x has a special hardware feature that loads a small program into internal memory at chip reset. This program can come from external PROM, the host port, or the link port, based on how the chip's external pins are hooked up. The loader uses this feature to load a program of arbitrary size into the ADSP-2106x's internal and external memory. The boot file generated by the loader can be tested in the ADSP-2106x Simulator.

### 6.1.1 Types Of Boot Files

The loader can be used to create four types of loadable images: the PROM-based loader, the host-based loader, the link port loader, and the specialized JTAG loader.

#### *6.1.1.1 PROM Booting*

The input to `LDR21K` is an executable file. From that file, the loader generates a PROM-based loader file (`*.ldr`) that can be programmed into a PROM chip. This file can be used as an input to most PROM/EPROM/EEPROM programmers.

Based on the requirements of your system design, you may need to change the `060_prom.asm` file. If so, here is how to recompile it (be sure to copy the original `060_prom.asm` file to your local directory, and only edit this copy of the original file):

```
asm21k -adsp21060 060_prom

ld21k -a 060_ldr.ach 060_prom
```

#### *6.1.1.2 Host Booting*

Command line switches `-bhost` and `-f` are used to generate a host-based loader: From the input file (`*.exe`), the loader generates a loader file (`*.ldr`) that can be used as an input or data file for a host based program that boots the processor through the data and address buses of the external port.

Based on the requirements of your system design, you may need to change the `060_host.asm` file. If so, here is how to recompile it (be sure to copy the original `060_host.asm` file to your local directory, and only edit this copy of the original file):

```
asm21k -adsp21060 060_host

ld21k -a 060_ldr.ach 060_host
```

#### 6.1.1.3 Link Port Booting

The LDR21K loader option -b*link* is used to generate an .ldr file that facilitates booting from an external source (such as a host processor) through the ADSP-2106x's link port buffer 4 using DMA channel 6. This .ldr file is loaded as a variable within the host's program and transmitted over the link port to load the booting processor. An example is included later in this chapter that illustrates the details of this process.

Based on the requirements of your system design, you may need to change the 060_link.asm file. If so, here is how to recompile it (be sure to copy the original 060_link.asm file to your local directory, and only edit this copy of the original file):

```
asm21k -adsp21060 060_link

ld21k -a 060_ldr.ach 060_link
```

#### 6.1.1.4 JTAG Booting (ADSP-21020)

The LDR21K loader may be used to download a small kernel of booting instructions that utilize the JTAG test access port (TAP) of the ADSP-21020 processor. The command line switch -bjtag is used to generate a boot file for the JTAG port. For complete information on JTAG booting of the ADSP-21020, see Chapter 10, *JTAG Downloader*, in the *ADSP-21000 Family Applications Handbook, Volume 1*. Sample source code and hardware design information are included.

Based on the requirements of your system design, you may need to change the 020_jtag.asm file. If so, here is how to recompile it (be sure to copy the original 020_jtag.asm file to your local directory, and only edit this copy of the original file):

```
asm21k  020_jtag

ld21k -a 020_jtag.ach 020_jtag
```

### 6.1.2 Distribution Files

For proper operation, the loader requires that the following files are located in the distribution directory of your disk.

In the ADI_DSP/21k/bin/ directory:

```
LDR21K
```

In the ADI_DSP/21k/etc/ directory:

```
060_ldr.ach
060_host.asm
060_host.exe
060_prom.asm
060_prom.exe
060_link.asm
060_link.exe

jtagboot.asm
jtagboot.ach
jtagboot.exe
020_jtag.asm
020_jtag.ach
020_jtag.exe
```

**Note**: The 060_ldr.ach file is a generic architecture file for PROM, host, or link port booting of all SHARC processors.

## 6.2 INVOKING THE LOADER

The loader is invoked with the LDR21K command. Any file that has been successfully processed by the linker can be used as input to the loader. Several options are available through the use of command line switches and additional files. The command line command structure is:

```
LDR21K infile  [-a filename  -b {type}   -c {custom}
                -e {segment} -f {format) -h
                -l filename  -o filename -v  -t N ]
```

**Implementation Notes:**

- Any executable (a C source program, assembly program, etc.) can be processed by the loader. The only requirement for an ADSP-2106x program to be processed is that it must not use address 0x020004 to store an instruction. The loader requires this location for last minute patching of the system state before starting the target executable. Your program should have a NOP or IDLE instruction at address 0x020004.

- The target application should be aware the DMAC6 control register on the ADSP-2106x will *not* equal zero. This is noted in the *SHARC User's Manual*.

- The loader takes the place of the MEM21K application. If you use the loader to initialize your target's memory, it is not necessary to use the MEM21K application. If you are using the C compiler, G21K, you should use the **-nomem** switch on the compilation line to stop the compiler from processing your executable with MEM21K.

- The input to LDR21K is an executable file generated by the Linker. Some of the boot loaders require particular segments to be defined. These requirements can be found in the section that discusses the types of loaders.

### 6.2.1 Command Line Switches

The loader supports the following command line switches:

- -a [*architecture filename* ]
  Specify an alternate architecture file. LDR21K follows the normal convention for default architecture file specification.

- -b[*type*]
  Where *type* indicates the method of booting to be used. The value of *type* can be **prom**, **host**, **link**, or **JTAG**. The default is -bprom .

- -c[*custom option* ]
  This switch causes the loader to change operation for a custom boot loader file. See the discussion on customizing your loader before using this option. Multiple -c switches may be used on the command line. The available options for *custom option* are **20004** and **20040**.

- -e[*segment name* ]
  Where *segment name* describes a segment in the architecture file that should be considered to contain 40-bit extended precision data. This switch will override a \WIDTH directive in the architecture description file.

- -f[*format*]
  Where *format* describes the output format of the .ldr file. The value of *format* can be *hex* for Intel hex records, *ASCII* for a 16-bit ASCII file, *include* for a file suitable to include as initialization data for a C program, or *binary* for a binary output file. The default is -fhex .

  The *binary* file format requires less space on a hard disk, and can be faster to read than the *include* or *ascii* formats. The binary format stores

48-bit words in three 16-bit blocks. The 16-bit blocks are written in the same order as all of the other formats. A typical code sequence used to access the binary file would be:

```
fd = fopen ("filename.ldr", READ_BINARY);
while (!feof (fd))
{
  fread (&short_data, 2, 1, fd);
  send_16bit_word_to_ADSP2106x (short_data);
}
```

- `-h`
  Help. Displays the command line syntax and a brief list of options.

- `-l [boot loader filename]`
  Specifies an alternate boot loader for this boot file. If this option is not specified, the loader will use the boot loader in the current directory (if available), or from the `ADI_DSP/21k/etc` subdirectory. See the discussion on customizing the boot loader before using this option.

- `-o [output filename]`
  Specifies an alternate output file. `LDR21K` uses the base of the input filename with the extension `.ldr` by default.

- `-v`
  Displays verbose output while running.

- `-t [N]`
  Specifies the maximum bus lock time. See the discussion below under "–t Switch".

### 6.2.2 -t Switch

There is a switch option of the loader which limits the number of cycles that the SHARC will spend on initializations. The switch is `-t` followed by a number (N). The number N is linearly related to the number of cycles that the SHARC will lock the bus during booting. Specifically, the number N instructs the SHARC to lock the bus for *about* 2∗N cycles at most. So, if you have a very fast host that does not like to be locked out, use a relatively small value for N; for example, `-t5`. N must be a positive, non-zero integer value greater than or equal to 4:

$N \geq 4$

Ideally, the value of N should be calculated from the amount of time the host (or bus) can tolerate a timeout. If the host can only tolerate 10 µsec of bus lock, N would be:

$$N = .75 \left[ \frac{\left( \frac{10\ \mu sec}{\left( \frac{1}{SS} \right)} \right)}{2} \right]$$

where SS is the speed of the processor in MHz and 1/SS is the processor cycle time. The .75 multiplier represents a 75% scaling factor.

### 6.2.3 Examples

**PROM-Based Loader**

The input to `LDR21K` is an executable file. Once you have created an executable file, the following command generates a PROM-based loader file (in the Intel Hex-32 file format):

`LDR21K` *`filename`*`.exe`

A loader file (*`filename`*`.ldr`) generated with this command line can be used as an input to most PROM programmers.

**Host-Based Loader**

Use the following command line to generate a host-based loader:

`LDR21K` *`filename`*`.exe -bhost -fascii`

The loader file (*`filename`*`.ldr`) can be used as an input or data file for a host based program.

**Link Port Loader**

To create a link port boot file for your system, invoke the loader with its `–blink` switch:

`LDR21K` *`filename`*`.exe –blink -fascii`

This output file (*`filename`*`.ldr`) can be used as a data file for the booting device connected to link buffer 4.

**JTAG Loader**
To create a JTAG boot file for your ADSP-21020 system, invoke the loader with its `-bJTAG` switch:

`LDR21K` *`filename`*`.exe -bJTAG`

This output file (*`filename`*`.ldr`) can be used as a data file for the booting device connected to the JTAG-compliant test access port (TAP).

## 6.3 SIMULATING LOADER FILES

The `wsim060` and `wsim020` simulators support the simulation of boot loading for both the host and PROM formats.

To simulate a loader file, it is necessary to modify one or more entries in the `wsim060.ini` (for the ADSP-2106x simulator) or `wsim020.ini` (for the ADSP-21020 simulator) files. Use a text editor to open the appropriate `*.ini` file and make the necessary modifications. In the startup section `[startup_sect]`, locate the following entry:

```
; "ldr_file=dev:\path\file" designates a loader file to boot upon
; startup.
; A "loader" file (*.ldr) is generated by the LDR21K.exe program
; and used for simulation of the DMA boot process.
; Specifying both an exe_file and a ldr_file generates an error.
;ldr_file=
```

Remove the ';' in front of the last line of this entry and add the name of the `*.ldr` file to the right of the '=' operator. Do not allow spaces between characters. Specify the full drive and pathway. For example:

```
ldr_file=c:\adi_dsp\21k\proj\exp.ldr
```

If an `ldr_file` **and** an `exe_file` are both specified in the `*.ini` file, an error message will be generated when the simulator is started up. The `exe_file` assignment statement must be removed from the `*.ini` file by adding a ';' as the first character in that line. This "comments out" the statement so that it is seen by the simulator program as a simple comment line, and thus ignored. To do this, locate the following entry:

```
; "exe_file=dev:\path\file" designates an executable file to load
; upon startup.
; Default is "21k.exe" in the "Working Directory" as specified for the
; wsim21k.exe program item by the File/Properties menu of
; the Windows Program Manager.
; Specifying both an exe_file and a ldr_file generates an error.

exe_file= c:\adi_dsp\21k\proj\cache.exe
```

Change the last line to:

```
;exe_fle= c:\adi_dsp\21k\proj\cache.exe
```

Then save the file and exit the text editor. When the simulator is started up, the specified `*.ldr` file is loaded into the simulator. The simulator determines which type of boot to simulate, and when the **Execute | Run/ Halt** menu function (or the F4 hotkey) is selected it immediately begins the boot process. You can interrupt the boot process by pressing the F4 hotkey. Booting can be resumed by reselecting the **Execute | Run/Halt** menu function (or the F4 hotkey).

## 6.4 CREATING A BOOT FILE

An ADSP-2106x executable program to be used with the loader must not have an instruction located at address 0x020004 in internal memory—the loader uses this location for last-minute patching of the system state before jumping to the restart vector. A `NOP` or `IDLE` instruction should be placed at this location. Remember also that DMA Channel 6 is used for EPROM, host, and link port booting, and the `DMAC6` control register is specially initialized for this purpose.

`LDR21K` uses a five-step process to create a boot file:

1. Parse the executable file.
2. Separate zero-valued initializations.
3. Write the specified boot loader to the boot file.
4. Write the memory initialization blocks to the boot file.
5. Write the final initialization (that overwrites the boot loader).

The following paragraphs describe each of these steps in detail. The boot file is always generated with the least significant values first.

### 6.4.1 ParsingThe Executable File

The loader reads the executable file to determine which segments are RAM based, and therefor need to be copied into the boot file. All segments that are declared as RAM in the architecture description file and contain initialization data are processed by the loader. The loader ignores segments that are declared in the architecture description file as ROM, and any segment that does not contain data.

The loader supports executables that have both RAM and ROM segments. The ROM segments are processed using the PROM splitter (`spl21k`) tool, while the RAM based segments are processed by the loader.

### 6.4.2 Separating Zero-Valued Initializations

Executables often contain large ranges of memory that are initialized to zero at system startup. These ranges can be efficiently represented as a single 48-bit word, rather than storing a long string of zeros, and can be performed more efficiently at boot time.

The RAM sections are scanned for runs of zero-valued initializations. These areas are then removed from the surrounding non-zero initializations and compressed.

### 6.4.3 Writing The Boot Loader

The boot loader contains instructions to read the rest of the boot file and initialize the system RAM. This means that the boot loader must be the first part of the boot file downloaded to the target. Therefore, the loader copies the boot loader to the beginning of the boot file.

The ADSP-2106x automatically boots 256 48-bit words from the appropriate device in all of its booting modes. The ADSP-21020 processor with JTAG boot support also reads 256 48-bit words from the boot PROM into the interrupt vector table. In most cases, the boot

loader is the default version supplied with the tools release in the `ADI_DSP/21k/etc` directory as executable files.

If there are particular hardware requirements or features in your application that need to be initialized before system memory, the boot loader can be customized for your requirements. See the sections later in this chapter discussing the appropriate boot loader for more details.

### 6.4.4 Writing Memory Initialization Blocks

The initialization data is written into the boot file following the boot loader. This is because the boot loader is responsible for initializing memory based on the specific boot format that the loader produces. The boot loaders for the ADSP-2106x initialize external memory by reading a word into internal memory, then writing that word to external memory.

When the boot loader attempts to access external memory, the boot loader sets the buslock bit (BUSLK bit of the MODE2 register) to lock the host off of the external bus. After the initialization values (or zero values) have been written to memory, the buslock bit is deasserted, allowing the host to write additional data to the processor.

**Note:** The period of time that the processor locks the bus can be limited using the `-t` switch.

Each initialization block is identified by a 48-bit tag which preceeds the block. Each type of initialization (e.g. zero 32 bit, initialize 40 bit ...) has a unique tag number. The tags for each type of initialization are listed in Table 6.1:

| *Tag Number* | *Initialization Type* |
|---|---|
| 0 | FINAL INIT |
| 1 | ZERO DM16 |
| 2 | ZERO DM32 |
| 3 | ZERO DM40 |
| 4 | INIT DM16 |
| 5 | INIT DM32 |
| 6 | INIT DM40 |
| 7 | ZERO PM16 |
| 8 | ZERO PM32 |
| 9 | ZERO PM40 |
| 10 | ZERO PM48 |
| 11 | INIT PM16 |
| 12 | INIT PM32 |
| 13 | INIT PM40 |
| 14 | INIT PM48 |

**Table 6.1 Memory Initialization Block Tag Numbers**

### 6.4.4.1 Zero-Valued Initialization Format

When the boot loader reads a tag whose value is 1, 2 ,3 7, 8, 9, or 10, it executes the appropriate initialization routine. A single 48-bit word following the tag is an address length pair indicating the starting address of the memory block in the upper 32-bits and the length of the initialization in the lower 16-bits.

An initialization of 16 or 32 bit memory (either PM, DM, Internal, or External) is done in a loop which writes a zero-value DREG to memory. Any initialization of 40 or 48-bit PM memory uses a write with the PX register set to zero. After the boot loader completes this initialization block, it reads the next tag and executes the appropriate initialization routine.

The loader will break a large initialization into two (or more) smaller initialization sequences if necessary. The loader does this when the length of the segment exceeds either 65535 (16-bit) words, or if it exceeds the value specified with the `-t` command line option.

### 6.4.4.2 Non-Zero Initialization Format

When the boot loader reads a tag whose value is 4, 5, 6, 11, 12, 13, or 14, it executes the appropriate routine to initialize memory to a non-zero value. A single 48-bit word following the tag is an address length pair indicating the starting address of the memory block in the upper 32-bits and the length of the initialization in the lower 16-bits.

The boot loader enters a loop which reads one 48-bit word and writes the appropriate width value to memory. This loop is repeated once for each word being initialized. After the boot loader reads each value from the host, link, or PROM, it locks the bus while it writes the value to memory. The bus is unlocked when the write is completed.

When the boot loader has finished initializing the specified memory block, it reads the next tag and executes the appropriate initialization routine.

The loader will break a large initialization into two (or more) smaller initialization sequences if necessary. The loader does this when the length of the segment exceeds either 65535 (16-bit) words, or if it exceeds the value specified with the `-t` command line option.

### 6.4.5 Writing The Final Initialization

The final initialization is indicated by a 0 value tag. When the loader detects this tag, it reads the next 48-bit word. This word indicates the instruction that should be located at 0x20040 when the loading is complete.

The next 256 48-bit words are loaded into memory over the boot loader. This process is executed with a DMA transfer sequence of 256 words which, when completed, `RTI` is executed at address 0x20040, returning to address 0x20004. An instruction at this address sets SYSCON to its initialization value, and writes the correct instruction at address 0x20040. The code then proceeds to address 0x20005 which should be the beginning of user application code.

Note that only the DMAC6 register has a different value than at the startup condition. The user application code is otherwise intact.

## 6.5 SPECIALIZED BOOT FILES

### 6.5.1 PROM Boot File For SHARC Clusters

The `LDR21K` loader allows clusters of SHARCs to be booted from a single PROM. For ADSP-2106x silicon earlier than revision 1.0, the BMS signals of all SHARCs must be ANDed together and tied to the boot PROM's chip select. For revision 1.0 silicon and later, all the SHARC's BMS signals may be tied directly together and to the PROM's chip select.

The loader accepts multiple executables in its command line. The first `.exe` in the loader command line will be loaded into the processor with ID=1, the second `.exe` will be loaded into the processor with ID=2, etc., up to six processors.

## 6.6 DETAILED EXAMPLES

### 6.6.1 EPROM Boot File

Creating an EPROM boot file for your system is the simplest type of boot file to make. Run the loader on your executable file with its `–bprom` switch:

`LDR21K` *`filename`*`.exe –bprom`

The output file, *filename*.ldr, can be used with a PROM programming device to create the boot EPROMs or EEPROMs for your system.

It may seem as if the Prom Splitter tool and the Loader tool are used for the same purpose. Actually, each tool has its own area of applications. In general, the PROM Splitter is used to create EPROMs in cases where 48-bit wide program memory (or even 32-bit wide data memory) external PROMs are used. All members of the ADSP-21000 Family could use the PROM Splitter in these circumstances.

On the other hand, the Loader is capable of creating boot code of any standard bit-width from byte-wide (8-bit) external EPROMs. Because of this, and the ADSP-2106x SHARC's more flexible booting options, the loader is the preferred tool.

### 6.6.2 Host Processor Boot File

To create a host processor boot file for your ADSP-2106x SHARC-based system, invoke the loader with its `-bhost` and `-fascii` command line switches:

`LDR21K` *`filename`*`.exe -b`*`host`* `-f`*`ascii`*

This output file, *filename*.ldr, can be used as an input or data file for the host processor's program to transmit to the SHARC processor at the proper time. The `-fascii` switch formats the output as a 16-bit ASCII file, since the 16-bit bus width will be used at the ADSP-2106x's external port.

When host booting is configured, the ADSP-2106x will enter slave mode after reset, waiting for the host processor to download the boot code. The host boot file created by the loader (*filename*.ldr) requires the host processor to perform the following sequence of actions:

1. The host initiates the booting operation by asserting the ADSP-2106x's HBR input.

2. After the host receives the HBG signal back from the ADSP-2106x, it should write three 16-bit words to the external port DMA buffer 0 (EPB0). (EPB0 corresponds to DMA channel 6.)

3. The host can continue to write 16-bit words to EPB0 until the entire program is booted.

This simple procedure can be used as long as the entire application will reside only in the ADSP-2106x's internal memory. If any portions of the application are planned to reside in external memory, however, the host processor must deassert HBR after writing each 48-bit instruction (in three 16-bit words). This allows the boot loader program (running on the ADSP-2106x) to gain control of the bus, lock it, and transfer the data to external memory. The boot loader will complete any necessary transfers and then unlock the bus, letting the host regain control. This protocol allows boot loading to occur with very little software or support on the part of the host.

To summarize this procedure, when some (or all) of the application program is to be stored in external RAM, the host should follow these steps:

1. The host asserts HBR.
2. After receiving HBG back, the host writes three 16-bit words to the EPB0 buffer.
3. The host deasserts HBR.
4. Continue back to Step 1 if more instructions are to be booted.

### 6.6.3 Link Port Booting

To create a link port boot file for your system, invoke the loader with its `-blink` switch:

```
LDR21K filename.exe -blink
```

This output file, *`filename`*`.ldr`, can be used as a data file for the booting device connected to link buffer 4.

Example link boot files are located in the `21k/etc/linkboot` subdirectory. This example is designed to run on the ADSP-2106x EZ-LAB board, but may be modified for any system architecture. For the purpose of the example, assume a system design that has a primary SHARC processor link booting a secondary SHARC processor. The primary processor transmits the boot code from its link ports to the link ports of the secondary processor.

1. Create the example file to be run on the secondary processor. In this case this file will be `lsrq.asm`, which uses a timer to cause an `LSRQ` interrupt on the primary processor and blinks an LED

attached to `flag0` (on the EZ-LAB board) every time the interrupt is received. Use `make1.bat` to produce the `lsrq.ldr` file.

2. Create the example program to run on the primary processor (the transmitting link port). You must incorporate a routine here (`blsrq.asm`) to transmit the secondary processor's program over the link. This transmitter routine is called before the primary processor begins to run its main program.

   The LOADLENGTH variable in the `blsrq.asm` file must be changed each time the size of the transmitted file changes. Read the comments within the program `blsrq.asm` for more details. The `blsrq.asm` example will blink the EZ-LAB's LED attached to `FLAG0` every time the interrupt is received. Use `make2.bat` to produce the .exe or .ldr file for prom booting of `blsrq.asm` into the primary processor.

## 6.7 THE BOOT PROCESS

The actual boot process is similar for all four boot modes, but each is slightly different.

The simplest form of booting uses a Boot EPROM connected to the external address / data ports of the processor. The Boot EPROM program is mapped so that the first instruction is located at a specific address for that processor; see the appropriate processor reference manual for details. Execution of the application program then proceeds normally.

The other three boot modes will typically require the use of the loader utility to accommodate the physical means of downloading the boot loader executable into the target processor. In general, the boot process begins by downloading the boot loader onto the target system. The boot loader then sets up the system as necessary, and begins to load initialization data. The boot loader initializes the entire system, with the exception of the area occupied by the loader itself. Once the boot loader has finished initializing the rest of the system, it needs to load over itself. This is the most difficult operation, and is unique to each boot mode.

# 6 Boot Loader

### 6.7.1 The ADSP-2106x PROM Loader

The PROM loader is used when the external boot PROM is connected to the ADSP-2106x. PROM booting is activated by setting the EBOOT pin to high and the LBOOT pin low. The BMS pin is the chip select line for the boot PROM in this mode.

DMA channel 6 is set up at reset with the following values:

| | |
|---|---|
| II | 0x20000 |
| IM | uninitialized |
| C | 0x100 |
| CP | uninitialized |
| GP | uninitialized |
| EI | uninitialized |
| DMAC6 | 0x02A1 |

The DMA is set up for external port DMA enable, DTYPE is selected for instructions and packing mode is forced to 8-to-48 bit packing. The packing order is LSW first. The IMASK register is set to allow DMA6 interrupts.

The IM and EM registers are uninitialized, but the boot hardware works as if IM is set to 0x100 and EM is set to 0x600.

The byte-wide boot PROM must be connected to data bus pins DATA(23:16). The lowest address pins of the ADSP-2106x should be connected to the PROM address bits. The PROM's chip select should be connected to BMS and its output enable should be connected to RD.

The WAIT register is initialized to generate 6 wait states for each PROM access.

This loader uses an external PROM to hold the program. The first 256 48-bit words in the PROM are loaded automatically by the ADSP-2106x. This code represents the boot loader executable.

The boot loader accesses the boot file in the PROM by toggling the BSO bit in the SYSCON register. When the BSO bit is active, only the boot PROM can be accessed as external memory. When the BSO bit is deactivated, the ADSP-2106x can only access external memory.

The internal memory is initialized, followed by the loader overwriting itself.

The DMA automatically uses an 8-to-48 bit packing mode when the BSO

bit is enabled. The BSO bit deselects the external memory and accesses the boot PROM.

The last step of the boot loader is to initialize the memory in which it resides.

### 6.7.2 The ADSP-2106x Host Loader

This loader assumes that there is an external host that will be sending information to the ADSP-2106x through the external port EP0. The DMA is placed in slave mode, so the host is responsible for transferring data to the ADSP-2106x.

When hardware reset occurs, the ADSP-2106x core enters the IDLE state at location 0x20004, and the DMA corresponding to channel 6 is initialized with the following values:

| | |
|---|---|
| DMAC6 | 0x00A1 |
| II6 | 0x020000 |
| C6 | 0x0100 |

The host must assert the host bus request pin to indicate that it wants to send data to the ADSP-2106x. After the host receives the host bus grant signal from the ADSP-2106x, it should write three 16-bit words to DMA buffer 0 (DMAB0 corresponds to DMA channel 6).

If any portion of the appliction program resides in external memory, the host should drop the host bus request signal after writing three words. The host *must* drop the host bus request signal for at least five ADSP-2106x instruction cycles. The host may assert the host bus request signal any time after five instruction cycles.

When the host drops the host bus request signal, the ADSP-2106x has the option of locking the bus. Once the bus is locked, the ADSP-2106x can transfer data to external memory. The ADSP-2106x completes any necessary transfers, and unlocks the bus.

This protocol allows booting to occur with very little software or support of the host.

The host must supply data on the DATA(31:16) pins, and the write address must be for DMAB0.

The DMA control register of the ADSP-2106x is set for DMA enabled, DTYPE selected for instructions, and packing mode set for 16-to-48 bits with LSW first.

# 6 Boot Loader

When the host boot is complete, the ADSP-2106x application is started. The application must be aware that the value in DMAC6 is non-zero, and that the IMASK is set to allow DMA6 interrupts.

## 6.8 LOADER EXECUTABLES

There are four different boot loader executables, one for each type of loader file generated by LDR21K. This section describes how each loader works, and how to customize the loaders for your application.

The sources for the boot loader executables are located in the library source directory. The library source directory is ADI_DSP/21k/lib/src. You should copy the source file to a working directory and only modify the working copy. Do not modify the source file.

There is an architecture file associated with the loader. The architecture file is necessary because each loader must be an executable file, rather than just an object file. The architecture file indicates what memory is used by the loader, and what memory is used for temporary storage.

The boot loader is a small assembly language program that initializes system memory, based the initialization data. The boot loaders are all 256 words long.

The following paragraphs describe the boot loaders in detail. You should refer to these paragraphs if you need to modify the loader for your application.

### 6.8.1 Prom Loader

060_prom.asm is the assembly source file for the ADSP-2106x PROM loader. The architecture file is named 060_ldr.ach.

060_prom.asm contains the boot loader that is loaded into seg_ldr at reset. The file begins by defining macros that contain addresses of various IOP registers that are used in the code.

The boot loader itself begins with part of the interrupt vector table of the ADSP-2106x. The table contains the interrupts vectors, up to and including

the low priority timer interrupt. Most of these interrupts are not necessary for the operation of the boot loader, but were included in case custom modifications required them.

The only interrupts used by the boot loader are the reset interrupt (0x020004 - 0x020007) and the DMA6 interrupt (0x020040 - 0x020043).

The boot loader begins by reading the current value of the `SYSCON` register and saving it. Three copies of `SYSCON` are used in the program; one that contains the original value of `SYSCON`, one that contains `SYSCON` with the `BSO` bit set, and a third with the `BSO` bit cleared.

The `IMASK` register is set to allow `DMA6` and `DMA7` interrupts, and the `MODE1` register is set to enable interrupts and allow nesting.

The boot loader uses a routine called `read_PROM_word` to fetch a single 48-bit word from the PROM. The boot loader reads a tag from the PROM to determine what type of initialization should be performed. The boot loader executes a small routine to initialize the memory as defined by the tag.

After a block of memory has been initialized, the boot loader jumps back and and reads the next tag from the PROM. When a tag of 0 is read, the loader executed the final initialization.

**Final Initialization**
The boot loader reads a word from the boot PROM. This 48-bit word represents the instruction that belongs at address 0x20040 for the target application. This instruction is loaded into the 48-bit PX register after the boot loader has finished initializing internal memory. The boot loader requires that address 0x020040 (the DMA6 vector address) contain an RTI instruction because a DMA6 interrupt is generated when the initialization is complete.

The R9 register is loaded with 0xb1db0000; this is the encoded instruction `PM(0,I8)=PX`.

The boot loader then reads in the next 256 words from the boot PROM. This data overwrites the loader itself.

### 6.8.2 Host Loader

`060_host.asm` is the source file for the ADSP-2106x Host loader.

`060_host.asm` contains the boot loader that is loaded into `seg_ldr` at reset. The file begins by defining macros that contain addresses of various IOP registers that are used in the code.

The boot loader itself begins with part of the interrupt vector table of the ADSP-2106x. The table contains the interrupts vectors, up to and including the low priority timer interrupt. Most of these interrupts are not necessary for the operation of the boot loader, but were included in case custom modifications required them.

The only interrupts used by the boot loader are the reset interrupt (0x020004 - 0x020007) and the DMA6 interrupt (0x020040 - 0x020043).

The boot loader begins by reading the current value of the `SYSCON` register and saving it. Three copies of `SYSCON` are used in the program; one that contains the original value of `SYSCON`, one that contains `SYSCON` with the `BSO` bit set, and a third with the `BSO` bit cleared.

The `IMASK` register is set to allow `DMA6` interrupts, and the `MODE1` register is set to enable interrupts and allow nesting.

The boot loader uses a routine called `read_PROM_word` to fetch a single 48-bit word from the host. The DMA6 interrupt vector sets the bus lock bit of the mode register, preventing host bus access. The boot loader reads a tag from the 48-bit word retrieved from the host to determine what type of initialization should be performed. The boot loader executes a small routine to initialize the memory as defined by

the tag. The bus lock is cleared and the boot loader if free to attemp reading another word from the host.

After a block of memory has been initialized, the boot loader jumps back and and reads the next tag from the host. When a tag of 0 is read, the loader executed the final initialization.

**Final Initialization**
The boot loader reads a word from the host. This 48-bit word represents the instruction that belongs at address 0x20040 for the target application. This instruction is loaded into the 48-bit PX register after the boot loader has finished initializing internal memory. The boot loader requires that address 0x020040 (the DMA6 vector address) contain an RTI instruction because a DMA6 interrupt is generated when the initialization is complete.

The R9 register is loaded with 0xb1db0000; this is the encoded instruction `PM(0,I8)=PX`. The contents of R9 are stored at address 0x20040. This instruction is executed and overwrites the proper instruction into location 0x20040.

### 6.8.3 Link Loader

`060_link.asm` is the source file for the ADSP-2106x Link loader.

### 6.8.4 JTAG Loader (ADSP-21020)

`020_jtag.asm` is the source file for the ADSP-21020 JTAG loader.

The architecture file looks like the following:

```
.SYSTEM  ADSP21020_JTAG_boot_strap_loader_Architecture;
.PROCESSOR = ADSP21020;
.SEGMENT/PM/RAM/BEGIN=0x00001000  /END=0x000001ff    seg_ldr;
.SEGMENT/DM/ROM/BEGIN=0x00800000  /END=0x00ffffff   boot_rom;
.ENDSYS;
```

`020_jtag.asm` contains the boot loader that is loaded into `seg_ldr` at reset. This segment is initialized by the jtagboot routine and is loaded over the JTAG TAP. The boot loader then reads the external PROM. Each time it reads from memory, it reads six 48-bit words. The control structure is similar to the `060_prom.asm` file described earlier.

The boot loader uses a routine called `read_PROM_word` to fetch a single 48-bit word from the boot PROM. The boot loader reads a tag from the 48-bit word retrieved from the boot PROM to determine what type of initialization should be performed. The boot loader executes a

small routine to initialize the memory as defined by the tag.

After a block of memory has been initialized, the boot loader jumps back and and reads the next tag from the host. When a tag of 0 is read, the loader executed the final initialization.

## 6.9 ERROR MESSAGES

LDR21K may generate the following runtime error messages:

```
ADI_DSP environment variable not defined
```

This message indicates that the environment variable ADI_DSP is not defined. This variable should point to the base of the distribution directory for the software tools. See the installation instructions.

```
Illegal flag ".." ignored
```

LDR21K encountered a switch that it did not expect. As the message indicates, the switch is ignored and processing continues.

```
Address 0x20004 should be a NOP or IDLE
```

The loader uses address 0x020004 to store an instruction which resets the processor to it's reset state after boot loading is complete. User code should not use address 0x20004 to store a valid instruction.

The loader replaces the instruction currently at address 0x020004 with the necessary instruction. If you application uses address 0x020004, you must customize the boot loader.

See the section on the host or PROM loader for more information on the use address 0x020004 by the loader.

```
Unknown loader section ".." encountered
```

This error indicates that an unknown section is included in the boot loader executable. The only segments that have any meaning in the boot loader executable are seg_tmp and seg_ldr, all other segments are ignored.

If you use a custom loader, be aware that all segments other than `seg_tmp` and `seg_ldr` are ignored.

If you are using one of the boot loader executables provided in the software releases, this probably indicates an installation error.

### `-a switch used without a file name`

This message indicates that an attempt was made to use the `-a` switch without a corresponding architecture file name. See the section on command line switches for more information.

### `-o switch used without a file name`

This message indicates that an attempt was made to use the `-o` switch without a corresponding output file name. See the section on command line switches for more information.

### `Unable to parse ".."`

This error occurs if the loader is unable to parse the architecture file. This error message is preceded by another message indicating the specific problem encountered.

### `Unable to read file header`

the loader is not the proper (COFF) file format. This error can occur when attempting to read the input executable to the loader, or when attempting to read the boot loader executable.

If this error occurs, make sure that you are using the correct file for input.

### `Unable to read optional header`

This error occurs if one of the input files to the loader is not the proper (COFF) file format. This error can occur when attempting to read the input executable to the loader, or when attempting to read the boot loader executable.

If this error occurs, make sure that you are using the correct file for input.

### `Unable to read section header`

This error occurs if one of the input files to the loader is not the proper (COFF) file format. This error can occur when attempting to read the input executable to the loader, or when attempting to read the boot loader executable.

If this error occurs, make sure that you are using the correct file for input.

### `Unable to open boot loader file ".."`

This error indicates that the loader was not able to open the specified boot loader executable. It may indicate an installation error, or that the boot loader executable specified on the command line is incorrect.

### `Improper start address for loader`

This error indicates that there is a problem in the format of the boot loader executable. The hardware boot sequence on the ADSP-2106x loads in 256 48-bit words starting at address 0x020000. The boot loader executable *must* begin at address 0x020000. These addresses can be filled with `NOPs`, but they must be filled.

If you are using a custom boot loader executable, you should edit, re-assemble, and re-link your boot loader. If you are using the standard loader provided with your software release, this error probably indicates an installation error.

### `Improper length for loader`

This error indicates that the boot loader executable is longer than 256 48-bit words. The hardware boot sequence on the ADSP-2106x loads in 256 48-bit words starting at address 0x020000. The boot loader executable must be less than or equal to 256 48-bit words in length or unexpected result may occur.



If you are using a custom boot loader executable, you should edit, re-